I/O Threads to Reduce Checkpoint Blocking for an Electromagnetics Solver on Blue Gene/P and Cray XK6

Jing Fu
Dept. of Computer Science
Rensselaer Poly. Inst.
Troy, NY 12180
fuj@cs.rpi.edu

Misun Min
MCS Division
Argonne National Laboratory
Argonne, IL 60439
mmin@mcs.anl.gov

Robert Latham
MCS Division
Argonne National Laboratory
Argonne, IL 60439
robl@mcs.anl.gov

Christopher D. Carothers
Dept. of Computer Science
Rensselaer Poly. Inst.
Troy, NY 12180
chrisc@cs.rpi.edu

ABSTRACT

Application-level checkpointing has been one of the most popular techniques to proactively deal with unexpected failures in supercomputers with hundreds of thousands of cores. Unfortunately, this approach results in heavy I/O load and often causes I/O bottlenecks in production runs. In this paper, we examine a new thread-based application-level checkpointing for a massively parallel electromagnetic solver system on the IBM Blue Gene/P at Argonne National Laboratory and the Cray XK6 at Oak Ridge National Laboratory. We discuss an I/O-thread based, application-level, two-phase I/O approach, called "threaded reduced-blocking I/O" (threaded rbIO), and compare it with a regular version of "reduced-blocking I/O" (rbIO) and a tuned MPI-IO collective approach (coIO). Our study shows that threaded rbIO can overlap the I/O latency with computation and achieve near-asynchronous checkpoint with an application-perceived I/O performance of over 70 GB/s (raw of 15 GB/s) and 50 GB/s (raw I/O bandwidth of 17 GB/s) on up to 32K processors of Intrepid and Jaguar, respectively. Compared with rbIO and coIO, the threading approach greatly improves the production performance of NekCEM on Blue Gene/P and Cray XK6 machines by significantly reducing the total simulation time from checkpoint blocking reduction. We also discuss the potential strength of this approach with the Scalable Checkpoint Restart library and on other full-featured operating systems such as the upcoming Blue Gene/Q.

Keywords

Parallel I/O, checkpointing, fault tolerance, MPI/Pthread

1. INTRODUCTION

As current leadership-class computing systems such as the IBM Blue Gene series [18] and Cray XK6 move closer to exascale capability, the number of processors increases to hundreds of thousands, and the failure probability rises correspondingly [8]. Already, the fidelity of scientific simulations running on these systems has reached an unprecedented level, and a large amount of checkpoint data is being generated for fault tolerance or postprocessing purposes. The gap between computing capability and the back-end storage system on these systems is enlarging, not only because technology advances in storage media (e.g., HDD) have not kept up with CPU advances, but also because of the complexity of the parallel I/O software stack between parallel applications and the storage media. The even higher concurrency expected in the exascale era will worsen this problem. Thus, development of scalable I/O approaches is crucial in order to help users better utilize the computing cycles allocated on massively parallel systems and achieve more productive science per compute cycle.

One of the current trends on the newest supercomputer hardware is that they share a high degree of resources, including memory and caches within nodes, network infrastructure between nodes, and a shared storage I/O system for the whole machine. For example, the Blue Gene/Q will feature the 64-bit PowerPC A2 with 16 cores and 4-way hyperthreading. In these many-core systems, simultaneous multithreading will likely be fully supported, and a potential performance boost can be achieved by exploiting those large nodes with a hybrid model (MPI + Pthread, e.g.).

The key contribution of this paper is a performance study of different parallel I/O approaches applied to application-level checkpointing for the production petascale electromagnetics solver NekCEM (Nekton for Computational Electro-Magnetics) [1] on both the Blue Gene/P and Cray XK6. In particular, we compare the application-level, two-phase reduced blocking I/O (rbIO) that uses an I/O thread (POSIX thread), with the regular rbIO and a well-tuned MPI-IO collective I/O approach (coIO). We demonstrate the performance advantage of threaded rbIO and analyze the I/O

speedup as well as application production speedup compared with collective I/O. We show that by using threaded rbIO, application users will see a near-asynchronous checkpiont with minimal overhead and considerable reductin in application production time at large scale.

This paper is organized as follows. In Section 2, we introduce the petascale application code NekCEM, used in our study. In Section 3, we discuss several parallel I/O approaches. In Section 4, we describe the Blue Gene/P and Cray XK6 system and analyze detailed experiment results from different approaches. In Section 5, we compare our approaches with related work in the literature. In Section 6, we give our conclusions and discuss some future work.

2. SOFTWARE AND I/O FILE FORMAT

NekCEM [1] is an Argonne-developed computational electromagnetics code based on a spectral-element discontinuous Galerkin discretization with explicit schemes for time-marching [6, 24]. It features body-fitting, curvilinear hexahedral elements that avoid stairstepping phenomena of traditional finite-difference time-domain methods [34] and yield minimal numerical dispersion because of the exponentially convergent high-order bases [16]. Tensor product bases of the one-dimensional Lagrange interpolation polynomials using the Gauss-Lobatto-Legendre grid points result in a diagonal mass matrix with no additional cost for mass matrix inversion, which makes the code highly efficient. The stiffness matrix is also a tensor product form of the one-dimensional differentiation matrix. The hexahedral elements allow efficient operator evaluation with memory access costs scaling as $O(n)$ and work scaling as $O(nN)$, where $n = E(N+1)^3$ is the total number of grid points, E is the number of elements, and N is the polynomial approximation order [9].

NekCEM uses the Message-Passing Interface (MPI) programming model [12, 11] for communication and the single program, multidata model [15] so that each processor independently executes a copy of the same program on distinct subsets of data. The discontinuous Galerkin approach based on domain decomposition requires communication only at the element faces (excluding the information of vertices and edges) between neighboring elements through a numerical flux [17]. The face values at the interfaces are saved in a single array for the six components of the electric field $E=(E_x, E_y, E_z)$ and the magnetic field $H=(H_x, H_y, H_z)$, for only one time communication at each time step between neighboring elements. This approach can reduce communication latency by a factor of 6, rather than saving the face values into six different arrays for each component of the fields.

NekCEM includes three tasks that are performed consecutively at run time: presetup, solver, and checkpointing. Presetup routines initialize processors, set compile-time data sizes, read run-time parameters and global mesh data from input files, distribute mesh data to each processor, and assign numbering for nodal points and coordinates for a geometry. Solver routines compute the spatial operator evaluation and time iterations. Checkpointing routines produce output files for the global field data.

NekCEM has two input data files, providing the information on global mesh data and global mapping for vertices including processor distribution for each element. For simplicity, data files are kept in global format so that users are not required to deal with mesh partition before compile/runs, with easier management for many different mesh configurations. Data files are read at the beginning, before the actual solver runs. Since the read operation occurs only once during the whole execution, our optimization focuses on the more frequent write operations.

NekCEM uses the *vtk legacy* format for an output file [20], where the master header includes the application name, file type (binary or ASCII), application type, gridpoint coordinates, cell numbering, and cell type. Every output file has a master header followed by *data blocks*. The data block contains the actual values of the field from NekCEM computation that are sorted mostly in the order of fields. In each data block, there is a header recording metadata such as data block size and field name. The *master header* typically specifies metadata information such as application name, version, local state list, and offset table.

In this paper, we apply optimal configurations of different checkpointing techniques to NekCEM solver that has been optimized on cache and register usage inside do-loops for spatial operator calculation and use of assembly-coded *dgemm* routines designed for small matrices. Currently, the weak scalability of the solver on the IBM Blue Gene/P *Intrepid* at Argonne Leadership Computing Facility (ALCF) is high, demonstrating ∼96% parallel efficiency with 2,132, grid points per processor (total number of grid points $n = 69$, 139, and 279 million on the number of processors $np = 32$K, 65K, and 131K, respectively). The weak scaling results for the Cray XK6 *Jaguar* at Oak Ridge Leadership Computing Facility (OLCF), are not as consistently flat as for Intrepid. Jaguar has a higher latency and faster clock, implying that the relative latency is about three times higher than on Intrepid. In addition, Jaguar has significant system noise that impairs the repeatability of timing tests. For example, the 60% efficiency observed for $np = 131{,}072$ processors, while not terrible, is also pessimistically low given that the efficiency is up to 96% at a different time of run.

Good scaling of NekCEM [25] with a low number of points per processor indicates that the code can be readily extended to future extreme-scale computing resources. For example, if one requires 10,000 points per core, then a 1-billion-point computation can effectively use at most 100,000 cores. In contrast, if one is able to efficiently use just 1,000 points per core, a million cores can be effectively used for the same problem, solving it in one-tenth the time, thus reducing opportunity costs. Our NekCEM scaling results that such a scenario is realistic.

3. PARALLEL I/O APPROACHES

In this paper, we discuss optimal configurations of three different parallel I/O approches that are extended from our previous works [13, 14]: collective I/O, reduced-blocking I/O, and threaded reduced-blocking I/O. An architecture diagram for each approach is given in Figure 1. On a fixed number of processors (***np***), each approach specifies the number of *group* (***ng***) processors that access the file system and the number of output files (***nf***) generated by those *writer*

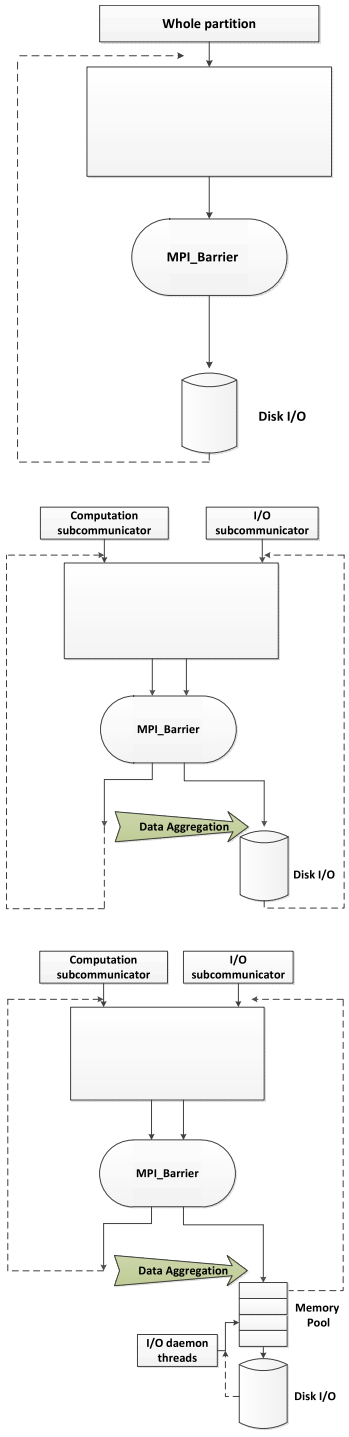

Figure 1: Architecture diagrams for different I/O approaches: coIO (top), rbIO (middle), and threaded rbIO (bottom).

processors.

Collective I/O (coIO) uses the MPI-IO library's built-in collective method to provide read/write operations to shared file descriptors with strided data access. It is relatively easy to implement and offers a few advantages over the traditional approach of 1 POSIX [19] file per process. In our implementation, all processors call the collective I/O routine to write data to a number of files. The number of output files, typically $\boldsymbol{nf}=2^m < \boldsymbol{np}$, m=0,1,2,..., is a user-tunable parameter. If $\boldsymbol{nf}$=1, all processors in `MPI_COMM_WORLD` use the `MPI_File_write_at_all_begin/end()` to write all data into one shared file. If $\boldsymbol{nf}$>1, the processors are divided evenly into $\boldsymbol{nf}$ ($=\boldsymbol{ng}$) groups, and the $\boldsymbol{np}/\boldsymbol{nf}$ processors in each group (i.e., local MPI communicator) collectively write to one file in parallel.

Reduced-blocking I/O (rbIO) divides a partition into application compute nodes (called *workers*) and I/O aggregator nodes (called *writers*). Thus $\boldsymbol{np}$ processors will be divided into $\boldsymbol{ng}$ groups with one *writer* per group. When the solver reaches a checkpoint phase, the workers send their data to the dedicated writer in its group (the *Data Aggregation* arrow as indicated in Figure 1 (middle)) through the P2P network of the system using `MPI_Isend()` and return. The writers aggregate the data, reorder data blocks, and write large data blocks to disk using collective or noncollective functions (depending on the number of files $\boldsymbol{nf}$ desired).

Threaded reduced-blocking I/O (*thread rbIO*) is similar to regular rbIO except that the writers spawn a thread to do the I/O write process in the background while the main MPI thread goes to the computation for next timestep as indicated in Figure 1 (bottom), instead of a blocking fashion in the regular rbIO. In this approach, both workers and writers can avoid waiting for disk I/O operations. Instead, the workers send the data to their corresponding writer and go to the next iteration computation. Meanwhile, the MPI thread of the writers aggregates data to the memory buffer and goes to the next iteration computation as well, in a relatively fast manner, leaving the I/O daemon thread to fetch the data from the buffer pool and scrabble the data to disk in the background.

4. PERFORMANCE AND ANALYSIS

In this section, we describe experiment test cases and demonstrate I/O performance of different I/O approaches that we implemented in NekCEM. We carry out performance tests using the GPFS [30] file system on the Blue Gene/P at ALCF and the Lustre file system on the Cray XK6 at OLCF.

4.1 System Overview

Intrepid, the Blue Gene/P system at ALCF, has 40,960 quad-core compute nodes (a total of 163,840 cores) and 80 TB of memory, with a theoretical peak performance of 557 teraflops. A 3D torus network serves as the P2P network for the compute nodes, with a bandwidth of 425 MB/s per direction [18]. In addition, a collective network connects compute nodes and I/O nodes, and a Gigabit Ethernet delivers data between I/O nodes and data storage servers. The BG/P runs a special lightweight kernel on the compute nodes called CNK (Compute Node Kernel). When a thread is created, it is allocated to the core with fewest threads running on that core. Since CNK does not support automatic thread switching on a core on a timed basis, we use BG/P's *dual* mode (2 MPI tasks per node) to allow our extra thread assigned to a core not running MPI task. I/O nodes (IONs) run a more full-featured kernel and act as a proxy between the compute node and the storage nodes to execute system

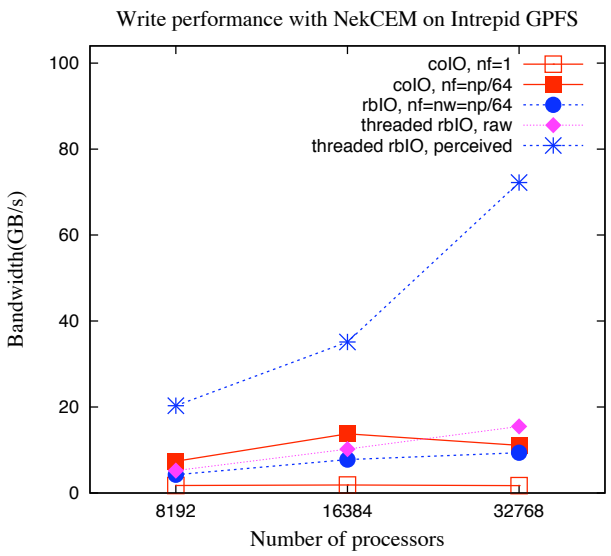

Figure 2: Write performance of different I/O approaches in NekCEM with GPFS on Intrepid, as a function of processor number for the problem sizes of (np, n, S) = (8K, 143M, 13GB), (16K, 275M, 26GB), and (32K, 550M, 52GB).

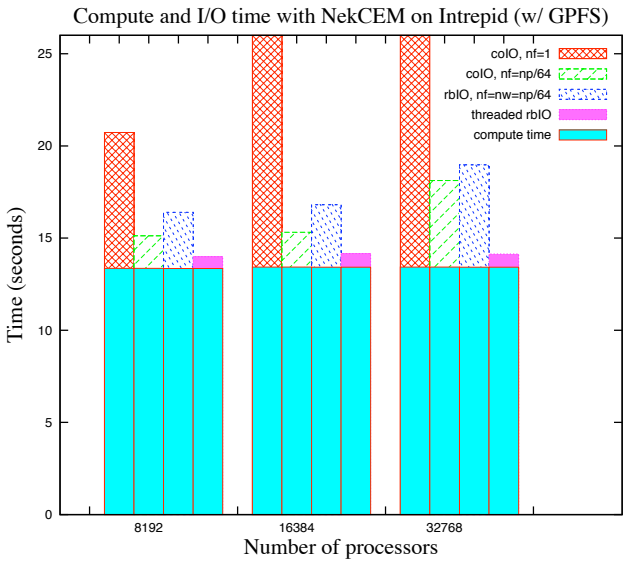

Figure 3: Compute and I/O time per checkpointing step for different I/O approaches in NekCEM with GPFS on Intrepid, as a function of processor number for the problem sizes of (np, n, S) = (8K, 143M, 13GB), (16K, 275M, 26GB), and (32K, 550M, 52GB). The checkpoint frequency is 1 per 100 computation steps.

calls and provide better scalability for the whole system. An ION and its compute nodes are referred to as a "pset." On Intrepid, each pset contains 1 ION and 64 compute nodes, with a total 640 IONs across the system.

The file system under the BG/P is a GPFS [30] and a PVFS [21] system, sharing 128 file servers that are connected to 16 Data Direct Network (DDN) 9900 SAN storage arrays. Each DDN exports the disk block as logical units, each directly connected to 8 file servers. A 10 Gigabit Ethernet connects the file servers to the I/O nodes. The theoretical peak bandwidth for read is 60 GB/s and that of write is 47 GB/s [21].

Jaguar, the Cray XK6 at OLCF, contains 18,688 compute nodes with a theoretical peak performance of 2.63 petaflops. Each compute node has an AMD Opteron "Interlagos" processor, which has 8 dual-cores units clocked at 2.2 GHz, and 32 GB of RAM. Each of the dual-core unit, called a "*module*", has two dedicated integer cores and two symmetrical 128-bit floating-point units that can be unified into one large 256-bit-wide unit. A hardware module is between a true dual-core that has two fully independent cores and a single-core processor that has simultaneous multithreading out of a single core. A *Gemini* high-speed interconnect router is shared by two nodes, and nodes within the compute partition are connected in a 3D torus. The system has a total of 299,008 cores and 598 TB of memory. Jaguar uses the Cray Linux environment, which consists of a Compute Node Linux microkernel on compute nodes and a full-featured Linux on login nodes.

The *spider* file system, Oak Ridge's main production file system, supports over 52,000 clients with 10 PB of disk space. It is the largest-scale Lustre file system in the world, with a peak bandwidth of 120 GB/s. Lustre is an object-based file system composed of three components: metadata servers, object storage servers (OSS), and clients. Each OSS manages one or more object storage targets (OSTs), and Spider has a total of 672 OSTs.

We note that the file systems at ALCF are shared by Intrepid, Eureka (a visualization system), and some other clusters whose I/O workload may affect the I/O performance observed on Intrepid. The Lustre file system at Oak Ridge is also shared between Jaguar and other systems such as Lens and Smoky, so the performance may vary by load from activity on other systems. In addition, all our tests were done under normal load, where there might be still system noise due to the traffic on the network from other online users.

4.2 Parallel I/O Performance for NekCEM

The test case is a 3D cylindrical waveguide simulation for different sizes of meshes and different numbers of processors with $(E, np) = (35K, 8K)$, $(68K, 16K)$, and $(137K, 32K)$. We used $N = 15$ so that the number of grid points per element is fixed at 16^3. The total number of fields in the checkpoint file is four. The output file sizes S per I/O step are (n, S) = (143M, 13GB), (275M, 26GB), and (550M, 52GB) on np = 8K, 16K, and 32K processors, respectively, with the checkpoint frequency one per 100 computation steps.

We measure the bandwidth as the total amount of data across all processors divided by the overall wall-clock time (including *open*, *write*, and *close* files) of the slowest processor to finish. The notion of *perceived* writing speed was defined as the speed at which worker processors can transfer

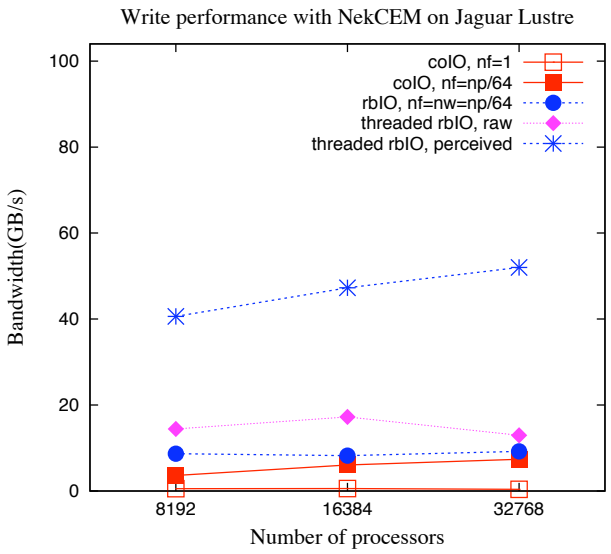

Figure 4: Write performance of different I/O approaches in NekCEM with Lustre on Jaguar, as a function of processor number for the problem sizes of (np, n, S) = (8K, 143M, 13GB), (16K, 275M, 26GB), and (32K, 550M, 52GB).

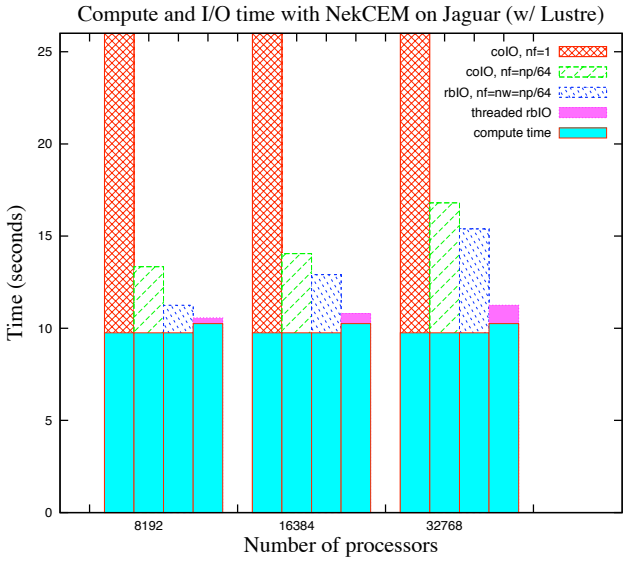

Figure 5: Compute and I/O time per checkpointing step for different I/O approaches in NekCEM with Lustre on Jaguar, as a function of processor number for the problem sizes of (np, n, S) = (8K, 143M, 13GB), (16K, 275M, 26GB), and (32K, 550M, 52GB). The checkpoint frequency is 1 checkpoint per 100 computation steps.

their data. This measure is calculated as the total amount of data sent across all workers over the maximum time of `MPI_Isend()` to complete. Most of these experiments were run multiple times, and the data points were sampled from the median number.

Figure 2 shows the write bandwidth of different approaches as a function of the number of processors on Intrepid. Here, the coIO with ***nf=1*** is at the bottom of the graph, showing the least bandwidth achieved compared with other approaches, because of the overhead of requiring all processors to communicate and synchronize during the collective write routine. The coIO approach with ***np:nf*** = *64:1*, where each collective write involves only 64 processors (with less communication and metadata operation overhead) and overall produces ***np/64*** files, achieves significantly better performance until a plummet at 32K processors, possibly due to noisy outliers in the partition [14]. By comparison, rbIO with ***np:nf*** = *64:1* provides decent raw performance in a more consistent manner. Moreover, because of BG/P's fast torus network, the workers can send their data block to corresponding writers in the group with `MPI_Isend()` (plus some other minor overhead) and go to the next iteration computation without any I/O blocking, thus achieving perceived speed bandwidth as high as 70 GB/s. At the same time, the I/O daemon thread can also achieve comparable raw bandwidth with regular rbIO and coIO and is able to finish the data-to-disk process before the next checkpoint phase arrives again.

Figure 3 shows the compute and I/O time per checkpoint step for different I/O strategies with NekCEM on Intrepid from 8K to 32K processors. The processor-to-file-count ratio (***np:nf***) for grouped coIO and rbIO was fixed at 64:1 because our previous study showed this to be a good practice [14]. As one can see from this figure, the compute time (for every 100 steps) is stable across different approaches from 8K processors to 32K processors. This stability is due to the excellent weak scaling property of NekCEM as described in Section 2. However, the I/O time varies widely depending on the approach. The coIO approach with ***nf=1*** spends more than 30 seconds (not fully shown in the figure) to finish 1 checkpoint, while the threaded rbIO approach spends less than 1 second. For threaded rbIO, since our compute thread and I/O thread run on different physical cores in the BG/P *dual* mode, the compute time was not affected by the I/O thread's activity. Since the I/O thread can achieve raw bandwidth similar to that of regular rbIO and coIO, the significant time difference of raw bandwidth and compute time ensures that there is little chance the I/O thread is still processing data when the next checkpoint arrives (this point is discussed further in the next section). Overall, threaded rbIO spends the least time on each compute+checkpoint phase and shows nearly perfect weak scaling from 8K processors to 32K processors on the Blue Gene/P.

Figure 4 shows the write bandwidth of different approaches as a function of the number of processors on Jaguar. Here the coIO with ***nf=1*** still achieve least bandwidth among others. The coIO with ***np:nf*** = *64:1* comes with consistently increasing bandwidth from 8K to 32K processors, topping at around 7 GB/s. On Jaguar, the coIO performs slightly worse than the rbIO, due to lack of collective buffering [36] and data block alignment optimizations [22] seen in ROMIO [35] implementation on the Blue Gene/P. The regular rbIO and threaded rbIO provide consistent performance at all scales, with over 17 GB/s raw performance at 16K

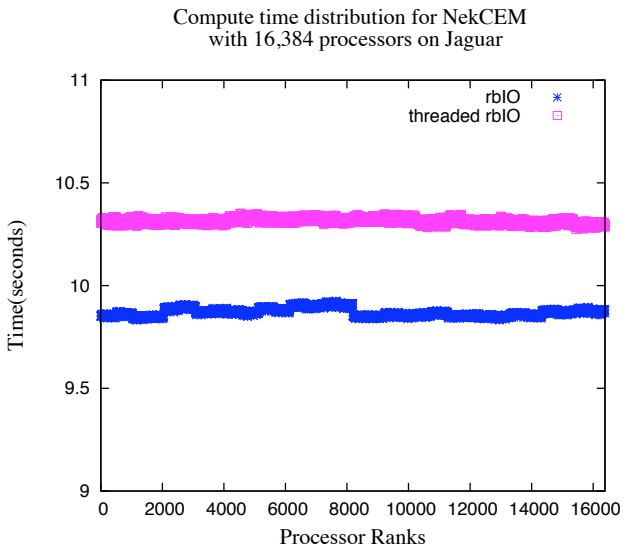

Figure 6: Compute time distribution (for 100 steps) among 16,384 processors for one checkpointing step, with nonthreaded and threaded rbIO in NekCEM on Jaguar.

processors. Here, the threaded rbIO has a decent performance advantage over regular rbIO due to the separation of data sending and data committing phases that benefit the computing of pure bandwidth. In addition, the perceived speed from threaded rbIO is consistently increasing from 40 GB/s to over 50 GB/s from 8K to 32K processors.

Figure 5 shows the compute and I/O time for different I/O strategies on Jaguar from 8K to 32K processors. The y-axis range is the same as in Figure 3. The coIO approach with ***nf=1*** performs worse than on BG/P (132 seconds for 1 checkpoint at 32K processors) because of a less-optimized implementation of the MPI-IO library. The regular rbIO finishes slightly faster than coIO at all scales. The threaded rbIO spends much less time on I/O than do the other approaches. However, because of the overhead of the thread switch on a core, the compute time for rbIO is slightly longer as well, as the higher bar for compute time on threaded rbIO indicates in Figure 5.

In Figure 6 we take a closer look at the compute distribution of threaded rbIO and nonthreaded rbIO at 16K processors on Jaguar. Because of the overhead from context switch of the MPI task and I/O thread, the MPI task on the writer processor lags behind a little and arrives at the synchronization point late for each computation step, thus slowing overall computation. The gap is about 0.5 second at all scales, which is much smaller than with threaded rbIO or other I/O approaches (normally >5 seconds).

4.3 Speedup Analysis

Previously we analyzed the theoretical speedup of time spent on I/O between rbIO and coIO [14]. Here we analyze the I/O speedup of threaded rbIO (denoted *trbIO*) over coIO by computing the overall time the application spends on I/O:

$$Speedup_{io} = \frac{T_{coIO}}{T_{trbIO}}. \quad (1)$$

The total time of all processors blocked by I/O operations, T_{coIO} and T_{trbIO}, can be defined by

$$T_{coIO} = np\frac{S}{BW_{coIO}}, \quad (2)$$

$$T_{trbIO} = np\frac{S}{BW_p}, \quad (3)$$

where BW_{coIO} and BW_p represent the bandwidths of coIO and perceived speed of threaded rbIO, respectively, and S is the file size. This leads to

$$Speedup_{io} = \frac{BW_p}{BW_{coIO}}. \quad (4)$$

On Jaguar, $Speedup_{io}$ is about 7–13× on different numbers of processors from 8K to 32K.

Next, we analyze the application production speedup of threaded rbIO with NekCEM versus coIO with NekCEM:

$$Speedup_{prod} = \frac{T_{coIO} + T_{comp}^{\text{coIO}}}{T_{trbIO} + T_{comp}^{\text{trbIO}}}. \quad (5)$$

Denoting the checkpoint frequency to be 1 checkpoint per f_{cp} computation steps, t_{comp} to be the time for single step computation, and X to be the number of computation steps that a checkpoint time equals to, one can express the production time speedup as

$$Speedup_{prod} = \frac{X_{coIO} * t_{comp}^{\text{coIO}} + f_{cp} * t_{comp}^{\text{coIO}}}{X_{trbIO} * t_{comp}^{\text{trbIO}} + f_{cp} * t_{comp}^{\text{trbIO}}}. \quad (6)$$

Suppose δ is the overhead of a single step computation with threaded rbIO compared with nonthreaded I/O (i.e., $t_{comp}^{\text{trbIO}} = (1+\delta) * t_{comp}^{\text{coIO}}$). Then we have

$$Speedup_{prod} = \frac{X_{coIO} + f_{cp}}{X_{trbIO} + f_{cp}} * \frac{1}{1+\delta}. \quad (7)$$

In our case on 32K processors of Jaguar, X_{coIO} is about 70× (of time for a single computation step), and X_{trbIO} is about 10×. Our f_{cp} has been 100, and δ is roughly 5%. Overall, then, the application speedup is roughly 50% on 32K processors of Jaguar.

One of the assumptions we are making here is that the total computation time T_{comp}^{trbIO} is always larger than or equal to the actual I/O time by adjusting the checkpoint frequency properly; otherwise the main computation task will be held till I/O thread finishes, a costly approach. On the other hand, given that threaded rbIO provides raw I/O bandwidth comparable to or even better than other approaches provide, if the application is generating more data than the I/O subsystem can take, threaded rbIO is still doing the best that can be done by hiding all the I/O latency possible.

If we adjust the checkpoint frequency higher (i.e., the f_{cp} becomes smaller), the speedup becomes more significant. The same applies if we increase the bandwidth of perceived speed by further reducing some overhead, or if we reduce the overhead of computation δ caused by the I/O thread on the same core.

5. RELATED WORK

Our previous work showed performance analysis for rbIO and coIO on Blue Gene/L and Blue Gene/P and parameter tuning practices [13, 14]. We demonstrated performance advantage of coIO and rbIO over traditional 1 POSIX file per processor approach and analyzed potential benefit of using rbIO on some platforms.

Seelam et al. [33, 31, 32] implement an application-level I/O caching system on the Blue Gene by analyzing I/O access pattern, prefetching requests and aggregating write-back data to storage. They use extra Pthreads to handle writing but it blocks the entire application thread when doing prefetching. They run several I/O benchmarks and report a 10% improvement on WRF execution time.

The I/O delegate and caching system (IODC) that is developed by Nisar et al. [7, 28, 29] uses a small percentage of compute nodes as dedicated delegates/aggregators to maintain a good file to I/O servers mapping and mitigate file block contention issues from independent I/O requests. This is implemented under MPI-IO layer and tested with application I/O kernels on up to 8K processors of a Cray XT4 system and other clusters. Our approaches introduced in this paper is implemented in user space and does not maintain an explicit mapping between aggregators and I/O file servers.

ADIOS by Lofstead et al. [23] is a portable metadata-rish I/O architecture that uses a set of lightweight, high-level API to tune applications for domain scientists with minimal effort and write data into a custom BP format to be converted back to scientific library like HDF5. ADIOS provides ease for generic application programmers while our work focuses on optimized performance for our application with the "bursty" I/O pattern.

Fahey et al. [10] performed I/O subsetting experiments on the Cray XT4 with four I/O approaches (MPI I/O, agg, ser, and swp) on up to 12K processors, and about 40% of peak write bandwidth was achieved. Lang et al. [21] used I/O benchmarks (BTIO, MADbench2, etc.) on the Blue Gene/P with up to 131K processors and reported I/O bandwidths of nearly 60 GB/s (read) and 47 GB/s (write) on PVFS. Borrill et al. [4, 5] used the MADbench2 benchmark on different systems (Lustre on Cray and GPFS on Blue Gene/L) to explore parameters such as concurrency, I/O library and output file number on up to 1K processors.

Pfeiffer et al. [2] use the MPI + Pthread hybrid model on RAxML phylogenetics code for production runs on SDSC clusters and conclude that hybrid code provide a 6× speedup compared to Pthreads-only code if tuned properly.

The Scalable Checkpoint Restart (SCR) library by Moody et al. [26] provides a multi-level checkpointing capability that can leverage local node storage in the form of RAM disk or SSD. They used pF3D benchmark on up to 8K cores and achieved a checkpoint speedup of 14× to 234× compared to a regular parallel file system. Our approach does not require special hardware support and uses user space RAM for data buffering. In the future, this gap may blur as the future supercomputer comes with a more full-featured OS that supports different hardware.

6. CONCLUSIONS AND FUTURE WORK

Our previous study [14] shows the benefit of using a tuned collective IO and user-level aggregation approach rbIO with a data-intensive scientifici application on Blue Gene/P. In this paper, we use rbIO with a separate I/O daemon thread so that the I/O work can be done asynchronously while the main computation tasks keep going into next iteration computation without being blocked by time-consuming disk I/O operations. From the application's perspective, it never sees any I/O in its way other than sending its data over to the aggregator, which can be done at a speed of more than 50 GB/s on a 32k processor partition on Jaguar with an actual raw bandwidth of 13 GB/s, which still prevails among other approaches. This approach is straight-forward to implement and works very well on Blue Gene/P and Cray XK6 machines, providing roughly 10× I/O improvement and 50% overall production time improvement over well-tuned MPI-IO collective approaches. In the fure, we plan to investigate into worker's computation overhead (δ) in threaded rbIO and test our approaches at larger scale.

Threaded rbIO uses extra user space RAM in compute node as buffer to absorb bursty I/O request from applications. This works well on existing machines and will work on any OS that has threading capability (including *dual* mode on BG/P and more full-featured OS such as the Blue Gene/Q). This principal coincides with design and simulation on future systems with HDD/SSD hybrid storage system such as burst buffer [27] in IOFSL [3].

7. ACKNOWLEDGMENTS

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357 while Jing Fu worked as a Givens Associate at the Mathematics and Computer Science Division at Argonne National Laboratory and under the subcontract 2F-30501. The authors acknowledge the Director's Discretionary resources and staff efforts provided by the Argonne Leadership Computing Facility and Oak Ridge Leadership Computing Facility.

8. REFERENCES

[1] NekCEM: Computational ElectroMagnetics Code. `http://wiki.mcs.anl.gov/nekcem/`.

[2] *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*. IEEE, 2010.

[3] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009.

[4] J. Borrill, L. Oliker, J. Shalf, and H. Shan. Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In *SC*, 2007.

[5] J. Borrill, L. Oliker, J. Shalf, H. Shan, and A. Uselton. HPC global file system performance analysis using a

scientific-application derived benchmark. *Parallel Computing*, 35(6):358–373, 2009.

[6] M. Carpenter and C. Kennedy. Fourth-order $2N$-storage Runge-Kutta schemes. *NASA Report TM 109112*, 1994.

[7] A. Choudhary, W. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham. Scalable I/O and analytics. *Journal of Physics: Conference Series*, 180(1), 2009.

[8] DARPA. DARPA ExaScale Software Study: Software Challenges in Extreme Scale System. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ecss%20report%20101909.pdf`, September 2009.

[9] M. Deville, P. Fischer, and E. Mund. *High-order methods for incompressible fluid flow,* Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2002.

[10] M. R. Fahey, J. M. Larkin, and J. Adams. I/O performance on a massively parallel Cray XT3/XT4. In *IPDPS*, pages 1–12, 2008.

[11] P. Fischer, J. Lottes, W. Pointer, and A. Siegel. Petascale algorithms for reactor hydrodynamics. *J. Phys. Conf. Series*, 125:012076, 2008.

[12] G. Fox, M. Hohnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on concurrent processors*. Prentice-Hall, Inc., Upper Saddle River, N.J., 1988.

[13] J. Fu, N. Liu, O. Sahni, K. Jansen, M. Shephard, and C. Carothers. Scalable parallel I/O alternatives for massively parallel partitioned solver systems. In *2010 IEEE International Symposium on Parallel and Distributed Processing*, April 2010.

[14] J. Fu, M. Min, R. Latham, and C. D. Carothers. Parallel I/O Performance for Application-Level Checkpointing on the Blue Gene/P System. In *CLUSTER*, pages 465–473, 2011.

[15] M. Heath. *The hypercube: A tutorial overview*. Hypercube Multiprocessors, SIAM Philadelphia, 1986.

[16] J. Hesthaven, S. Gottlieb, and D. Gottlieb. *Spectral methods for time-dependent problems,* Volume 21 of Cambridge monographs on applied and computational mathematics. Cambridge University Press, 2007.

[17] J. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods, algorithms, analysis, and applications*. Springer, 2008.

[18] IBM Blue Gene team. Overview of the IBM Blue Gene/P project. IBM Journal of Research and Development Volume 52 Issue 1/2, 2008.

[19] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–Part 1: System application program interface (API) [C language], 1996 edition.

[20] Kitware. *VTK User's Guide*. Kitware, Inc. 11th Edition, 2010.

[21] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of Supercomputing*, November 2009.

[22] W. Liao and A. Choudhary. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, Texas*, November 2008.

[23] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *IPDPS 09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2009.

[24] M. S. Min and P. Fischer. An efficient high-order time integration method for spectral-element discontinuous Galerkin simulations in electromagnetics. MCS, ANL, Preprint ANL/MCS-P1802-1010, 2011.

[25] M. S. Min and J. Fu. Performance analysis on the IBM BG/P for the spectral-element discontinuous Galerkin method for electromagnetic modeling. MCS, ANL, Preprint ANL/MCS-P1802-1010, 2010.

[26] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans*, November 2010.

[27] L. Ning, C. Jason, C. Philip, C. Christopher, R. Robert, G. Gary, C. Adam, and M. Carlos. On the role of burst buffers in leadership-class storage systems. In *MSST/SNAPI*, 2012.

[28] A. Nisar, W. keng Liao, and A. N. Choudhary. Delegation-Based I/O Mechanism for High Performance Computing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):271–279, 2012.

[29] A. Nisar, W. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[30] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[31] S. Seelam, I. Chung, J. Bauer, and H. Wen. Masking i/o latency using application level i/o caching and prefetching on blue gene systems. In *IPDPS*, 2010.

[32] S. Seelam, I.-H. Chung, J. Bauer, H. Yu, and H.-F. Wen. Application level i/o caching on blue gene/p systems. In *IPDPS*, pages 1–8, 2009.

[33] S. Seelam, I.-H. Chung, D.-Y. Hong, H.-F. Wen, and H. Yu. Early experiences in application level i/o tracing on blue gene systems. In *IPDPS*, 2008.

[34] A. Taflove and S. Hagness. *Computational Electrodynamics, The Finite Difference Time Domain Method*. Artech House, Norwood, MA, 2000.

[35] R. Thakur, E. Lusk, and W. Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[36] H. Yu, R. K. Sahoo, C. Howson, G. Almasi, J. G. Castanos, M. Gupta, J. E. Moreira, J. J. Parker, T. E. Engelsiepen, R. Ross, R. Thakur, R. Latham, and W. D. Gropp. High performance file I/O for the BlueGene/L supercomputer. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, February 2006.

The following paragraph should be deleted before the paper is published: The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.